APPENDIX A: SOURCE CODE LISTINGS

```
/*
The AbstractCommandTag class, which demonstrates:

1.  Conditional execution of the execute() method based on HTTP request
    method (GET, POST, or the like).

2.  Conditional execution of the execute() method based on the value of the
    HTTP request parameter named "key".  If this request parameter matches the
    descendant tag's attribute named "key", the execute() method is run.

3.  Page routing based on the success or failure of the execute() method.
    If the execute() method does not throw an exception, it has succeeded.  If
    the execute() method succeeds, the request is routed to the page specified
    in the tag's "onSuccess" attribute, if present.  If the execute() method
    fails, the request is routed to the page specified in the tag's "onFailure"
    attribute, if present.  An additional tag attribute named "redirect"
    controls whether this routing is performed inside the servlet engine
    (redirect='false') or through an HTTP redirect return code
    (redirect='true').

4.  Utility method which descendant classes can use to check for required
    HTTP request parameters in a standard manner.

5.  Utility method which can populate a JavaBean with the values contained
    in an HTTP request.

6.  Utility method which can set arbitrary properties in a JavaBean.

*/

package com.lightsurf.taglib;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.TagSupport;
import java.beans.*;
import java.lang.reflect.*;
import java.util.Enumeration;
import java.io.IOException;
import java.io.PrintWriter;
```

```
import java.text.*;
import java.util.*;

import com.lightsurf.util.*;

/**
 * AbstractCommandTag is the base class for all
 * command tags in the tag library framework.  This
 * tag provides the following services:
 * <ul>
 * <li>filter command by HTTP request method
 * <li>filter command by request parameter <code>key</code>
 * <li>redirect request on command success or failure using
 * either JSP forward or HTTP redirect
 * <li>set bean properties from request parameters using
 * introspection
 * </ul>
 * Descendant classes implement the <code>execute()</code>
 * method.  The class specifically does not have non-final
 * static members to increase its extensibility.
 *
 * @author Paul Egli
 * @version 1.0
 */
public abstract class AbstractCommandTag extends TagSupport {

    private static final int ABSTRACTCOMMANDTAG_START = 9600;
    private static final int SERVLETEXCEPTION = 9600;
    private static final int BEAN_ERROR       = 9601;

    public static String className = AbstractCommandTag.class.getName();

    public static final String KEY_PARAMETER  = "key";
    public static final String PAGE_PARAMETER = "page";
    public static final String METHOD_ANY     = "any";

    private String method = METHOD_ANY;
    private String key = null;
    private String onSuccess = null;
    private String onFailure = null;
    private boolean redirect = false;

    static{
        LogCodes.checkLogCode(className,ABSTRACTCOMMANDTAG_START,
                              LogCodes.TAGLIB_ABSTRACTCOMMANDTAG);
    }

    /**
     * Conditionally execute the command on GET, POST, or
     * ANY (default) HTTP method.
     *
     * @attribute method
     * @param method GET, POST, HEAD, etc., or ANY for any method
     */
    public final void setMethod(String method) {
        this.method = method;
    }
```

```
        /**
         * Getter for tag attribute <code>method</code>
         *
         * @return HTTP method to filter command on or ANY for no filtering
         */
        public final String getMethod() {
            return method;
        }

        /**
         * Optional command tag filter.  The command tag will
         * not execute if the request parameter named "key"
         * is missing or its value does not match the value
         * of this attribute.
         *
         * @attribute key
         * @param key    arbitrary string that must match the value of the
         *               request parameter named "key".
         */
        public final void setKey(String key) {
            this.key = key;
        }

        /**
         * Getter for the tag attribute <code>key</code>
         *
         * @return Arbitrary string set in the tag attribute as a
         *         request filter.
         */
        public final String getKey() {
            return key;
        }

        /**
         * Route to this URI on command success.
         *
         * @attribute onSuccess
         * @param onSuccess String containing an absolute or relative URL to
         *                  redirect or forward the request to if the
         *                  <code>execute()</code> method does not throw an
         *                  exception or throw an exception to the page
    exception
         *                  stack.
         */
        public final void setOnSuccess(String onSuccess) {
            this.onSuccess = onSuccess;
        }

        /**
         * Getter for the tag attribute <code>onSuccess</code>.
         *
         * @return String containing a URL to forward or redirect to
         *         if the <code>execute()</code> method succeeds.
         */
        public final String getOnSuccess() {
            return onSuccess;
        }
```

```
    /**
     * Route to this URI on command failure
     *
     * @attribute onFailure
     * @param onFailure String containing an absolute or relative URL to
     *                  redirect or forward the request to if the
     *                  <code>execute()</code> method throws an
     *                  exception or puts an exception to the page
exception
     *                  stack.
     */
    public final void setOnFailure(String onFailure) {
        this.onFailure = onFailure;
    }

    /**
     * Getter for the tag attribute <code>onFailure</code>.
     *
     * @return String containing a URL to forward or redirect to
     *         if the <code>execute()</code> method fails.
     */
    public final String getOnFailure() {
        return onFailure;
    }

    /**
     * If true, issue an HTTP 302 Redirect for onSuccess
     * or onFailure. If false, use JSP forward.  Default
     * value is false for redirect.
     *
     * @attribute redirect
     * @param redirect if true, use HTTP redirect.  if false, use JSP
forwarding
     */
    public final void setRedirect(boolean redirect) {
        this.redirect = redirect;
    }

    /**
     *
     * @return True if using HTTP redirect, false if using JSP forwarding.
     *         Default value is false for redirect.
     */
    public final boolean getRedirect() {
        return redirect;
    }


    /**
     * Main method that is overridden in descendant classes.
     * Any exceptions thrown by this method are caught and
     * added to the page exception queue.
     *
     * @param request Incoming servlet request.  HttpServletRequest is used
     *                so the command tag has access to the HTTP request
     *                method (GET, POST, etc) for selective execution.
     * @exception CommandTagException
     *                  Any runtime exceptions should be wrapped into
```

```
     *                    CommandTagException so they can be stored in the
     *                    page exception stack.
     */
    protected abstract void execute(PageContext pageContext) throws
CommandTagException;

    protected abstract boolean verify(PageContext pageContext) throws
JspTagException;

    /**
     * Calls the execute() method if the request method
     * is correct and the optional key parameter is
     * present in the request.  Puts CommandTagExceptions into the
     * exception queue for later retreival.
     *
     * @return
     * @exception JspException
     */
    public int doStartTag() throws JspException {
        HttpServletRequest request = (HttpServletRequest)
pageContext.getRequest();

        // IF there are errors then dont run any of the commands on this
page.
        if (TagFrameworkSupport.haveExceptionErrors(pageContext)){
            return SKIP_BODY;
        }

        // skip the execution if the key tag is missing
        if (key != null &&
!key.equalsIgnoreCase(KeyUtil.decodeForm(request.getParameter(KEY_PARAMETER
))))) {
            return SKIP_BODY;
        }

        // skip the command if the request method is not "any" or doesn't
matche the method parameter
        if (!method.equalsIgnoreCase(METHOD_ANY) &&
!method.equalsIgnoreCase(request.getMethod())) {
            return SKIP_BODY;
        }

        if (verify(pageContext)) {
            // perform form submission only if form validates successfully

            try {
                execute(pageContext);
            }
            catch (CommandTagException e) {
                // something dreadful happened.  throw an exception to the
page
                throw new JspTagException(e.getMessage());
            }

        }

        if (TagFrameworkSupport.haveExceptionErrors(pageContext)) {
            // redirect the page on success if the next attribute is set
```

```
            if (onFailure != null) {
                HttpServletResponse response = (HttpServletResponse)
pageContext.getResponse();
                if (redirect) {
                    try {

response.sendRedirect(response.encodeRedirectURL(onFailure));
                    }
                    catch (IOException e) {
                        throw new JspTagException("IOException while trying
to sendRedirect("+onFailure+") in AbstractCommandTag.doStartTag()");
                    }
                }
                else {
                    try {
                        RequestDispatcher rd =
pageContext.getServletContext().getRequestDispatcher(response.encodeURL(onF
ailure));
                        rd.forward(pageContext.getRequest(),
pageContext.getResponse());
                    }
                    catch (IOException e) {
                        throw new JspTagException("IOException while trying
to forward("+onFailure+") in AbstractCommandTag");
                    }
                    catch (ServletException e) {
                        throw new JspTagException("ServletException while
trying to forward("+onFailure+") in AbstractCommandTag");
                    }
                }
            }
        }
        // redirect the page on success if the next attribute is set
        else if (onSuccess != null) {
            HttpServletResponse response = (HttpServletResponse)
pageContext.getResponse();
            if (redirect) {
                try {

response.sendRedirect(response.encodeRedirectURL(onSuccess));
                }
                catch (IOException e) {
                    throw new JspTagException("IOException while trying to
sendRedirect("+onSuccess+") in AbstractCommandTag.doStartTag()");
                }
            }
            else {
                try {
                    RequestDispatcher rd =
pageContext.getServletContext().getRequestDispatcher(response.encodeURL(onS
uccess));
                    rd.forward(pageContext.getRequest(),
pageContext.getResponse());
                }
                catch (IOException e) {
                    throw new JspTagException("IOException while trying to
forward("+onSuccess+") in AbstractCommandTag");
```

```
                    }
                    catch (ServletException e) {
                        Log.printError(className,SERVLETEXCEPTION,
                                       "ServletException while trying to
5       forward(" + onSuccess + ")",
                                       (Throwable)e);
                        throw new JspTagException("ServletException while
        trying to forward("+onSuccess+") in AbstractCommandTag");
                    }
10              }
            }
            return SKIP_BODY;
        }

15

        /**
         * Utility method which throws a populated MissingParameterException
         * if any of the properties named in requiredParams
         * are missing from the servlet request
20       *
         * @param request
         * @param requiredParams
         * @exception MissingParameterException
         */
25      protected void checkForRequiredParameters(PageContext pageContext,
        String[] requiredParams) throws MissingParameterException {

            if (requiredParams == null) {
                return;
30          }

            ServletRequest request = pageContext.getRequest();

            boolean[] flags = new boolean[requiredParams.length];
35          for (int i=0; i < flags.length; i++) {
                flags[i] = false;
            }

            for (int i=0; i < requiredParams.length; i++) {
40              String param = requiredParams[i];
                String subparam = requiredParams[i];
                boolean multiple = false;
                if (param.endsWith("*")) {
                    subparam = param.substring(0, subparam.length()-1);
45                  multiple = true;
                }

                for (java.util.Enumeration e = request.getParameterNames();
        e.hasMoreElements();) {
50                  String name = (String) e.nextElement();
                    if (multiple) {
                        flags[i] = name.startsWith(subparam) &&
        (request.getParameter(name).length() > 0);
                    }
55                  else {
                        flags[i] = name.equals(param) &&
        (request.getParameter(name).length() > 0);
                    }
```

```
                    if (flags[i]) {
                        break;
                    }
                }
            }

            // construct MPEs and put into page
            for (int i=0; i < flags.length; i++) {
                if (!flags[i]) {
                    TagFrameworkSupport.throwToPage(pageContext, new
MissingParameterException("Required parameter " + requiredParams[i] + " is
missing", requiredParams[i]));
                }
            }
        }

        /**
         * Convenience method to set all bean properties from
         * request parameters.  Request parameters that do
         * not map to bean properties are ignored.
         * <p>
         * <b>Important note!</b>  Call this method with the
         * highest base class of the bean possible.  For example,
         * if a class or interface named Album extends a class
         * or interface named Container, call this method with
         * Container.class, not Album.class.  If you use Album.class,
         * this method will only set the object properties in
         * the Album class and will not look up to the parent
         * class for properties.
         *
         * @param cls     Class of the bean whose parameters are being set
         * @param bean    The JavaBean whose parameters are being set.
         * @param request The request containing parameters
         * @exception CommandTagException
         *                  Thrown if any introspection errors occur
         */
        public static void setBeanProperties(Class cls, Object bean,
ServletRequest request) throws CommandTagException {
            String propertyName = null;

            try {
                BeanInfo beanInfo = Introspector.getBeanInfo(cls);
                PropertyDescriptor[] descriptors =
beanInfo.getPropertyDescriptors();

                for (Enumeration e = request.getParameterNames();
e.hasMoreElements();) {
                    propertyName = (String) e.nextElement();
                    Method setter = null;

                    for (int i=0; i < descriptors.length; i++) {
                        PropertyDescriptor descriptor = descriptors[i];
                        if (descriptor.getName().equals(propertyName)) {
                            setter = descriptor.getWriteMethod();
                            if (setter != null) {
                                try {
                                    setter.invoke(bean,
coerceRequestParameterValue(setter, request.getParameter(propertyName)));
```

```
                                }
                                catch (IllegalArgumentException ex) {
                                    Log.printError(className,BEAN_ERROR,
                                                                "Could not set
property '" + propertyName + "' of bean '" +
                                                                cls.getName() + "'
to '" + request.getParameter(propertyName) +
                                                                ": " +
ex.getMessage());
                                }
                            }
                            // ignore read-only properties that were specified
in the request
                            break;
                        }
                    }
                }
            }
            catch (IntrospectionException e) {
                throw new CommandTagException(e.getMessage());
            }
            catch (InvocationTargetException e) {
                throw new CommandTagException("Unable to access property
'"+propertyName+"' of bean '"+cls.getName()+"' in AbstractCommandTag");
            }
            catch (IllegalAccessException e) {
                throw new CommandTagException("Unable to access property
'"+propertyName+"' of bean '"+cls.getName()+"' in AbstractCommandTag");
            }
        }

        /**
         * This routine is
         * used inside the <code>setBeanProperty()</code> and
         * <code>setBeanProperties()</code> methods to
         * automagically convert a String into the appropriate
         * type for an Introspection method call.
         *
         * @param setmethod The method passed in is examined for property type
         * @param value     The value to be converted to the appropriate type
for the method passed in.
         * @return an Object array containing the appropriate
         *         object type for a given method.
         * @exception IllegalArgumentException
         */
        private static Object[] coerceRequestParameterValue(Method setmethod,
String value) throws IllegalArgumentException {
            Class[] types = setmethod.getParameterTypes();
            if (types != null && types.length == 1) {
                if (types[0] == Boolean.TYPE) {
                    return new Object[] { new Boolean(value)};
                }
                else if (types[0] == Character.TYPE) {
                    if (value != null) {
                        return new Object[] { new Character(value.charAt(0))};
                    }
                }
                else if (types[0] == Byte.TYPE) {
```

```
                    return new Object[] { new Byte(value)};
                }
                else if (types[0] == Short.TYPE) {
                    return new Object[] { new Short(value)};
                }
                else if (types[0] == Integer.TYPE) {
                    return new Object[] { new Integer(value)};
                }
                else if (types[0] == Long.TYPE) {
                    return new Object[] { new Long(value)};
                }
                else if (types[0] == Float.TYPE) {
                    return new Object[] { new Float(value)};
                }
                else if (types[0] == Double.TYPE) {
                    return new Object[] { new Double(value)};
                }
                else if (types[0] == Date.class || types[0] ==
java.sql.Date.class) {
                    try {
                        return new Object[] {
TagFrameworkSupport.DATE_FORMAT.parse(value)};
                    }
                    catch (java.text.ParseException e) {
                        throw new IllegalArgumentException(e.getMessage());
                    }
                }
            }
            return new Object[] { value};  // default action
        }

        /**
         * Uses introspection to set a single property in a bean
         *
         * @param cls     The class of the bean to be read
         * @param bean    The object in which to set the property
         * @param propertyName
         *                the name of the property to read
         * @param propertyValue
         *                The value of the property as a String
         * @exception JspTagException
         *                    Thrown if the named property isn't present in the
         *                    bean.  Additionally, any problems with
introspection
         *                    are wrapped in a JspTagException
         */
        public static void setBeanProperty(Class cls, Object bean, String
propertyName, String propertyValue) throws JspTagException {
            try {
                BeanInfo beanInfo = Introspector.getBeanInfo(cls);
                PropertyDescriptor[] descriptors =
beanInfo.getPropertyDescriptors();
                Method setter = null;
                for (int i=0; i < descriptors.length; i++) {
                    PropertyDescriptor descriptor = descriptors[i];
                    if (descriptor.getName().equals(propertyName)) {
                        setter = descriptor.getWriteMethod();
                        break;
```

```
                }
            }
            if (setter == null) {
                throw new JspTagException(cls.getName()+" does not have a
property named '"+propertyName+"'");
            }

            Object result = setter.invoke(bean,
coerceRequestParameterValue(setter, propertyValue));
        }
        catch (IntrospectionException e) {
            throw new JspTagException(e.getMessage());
        }
        catch (InvocationTargetException e) {
            throw new JspTagException("Unable to access property
'"+propertyName+"' of bean '"+cls.getName()+"' in AbstractCommandTag");
        }
        catch (IllegalAccessException e) {
            throw new JspTagException("Unable to access property
'"+propertyName+"' of bean '"+cls.getName()+"' in AbstractCommandTag");
        }
    }

}
```

```
/*
TagFrameworkSupportExample.java contains methods which are called to store
and retrieve recoverable errors that may occur during the execution of
AbstractCommandTag descendant classes.
*/

/**
 * Class for static utility methods used
 * in the tag library framework.
 *
 * Methods which support error recording from AbstractCommandTag are
 * implemented in this class.
 */
public abstract class TagFrameworkSupport {

    /**
     * Adds an exception to the exception hold.  This method
     * is used to stack up exceptions for later processing
     * instead of throwing exceptions to the servlet
     * container.  Descendants of AbstractCommandTag can use
     * this method to indicate that a recoverable exception
     * has occurred during processing of their execute()
     * methods.
     *
     * @param pageContext
     * @param exception
     */
    public static void throwToPage(PageContext pageContext,
CommandTagException exception) {
        if (pageContext == null) {
            return;
        }

        ServletRequest request = pageContext.getRequest();

        Vector exceptionHold = (Vector)
request.getAttribute(TagConstants.EXCEPTION_HOLD_NAME);
        if (exceptionHold == null) {
            exceptionHold = new Vector(0, 1);
            request.setAttribute(TagConstants.EXCEPTION_HOLD_NAME,
exceptionHold);
        }

        exceptionHold.addElement(exception);
    }

    /**
     * Examines the page exception stack and returns an
     * array of exceptions which match the class type
     * passed in.
     *
     * @param pageContext
     *              JSP page context used to retrieve the exception stack
     * @param type   name of the exception class to retrieve.
     * @return an array of CommandTagExceptions.  Will return a zero-
     *         length array if no exceptions in the stack matched the
     *         given type or if the type parameter is null.
     * @exception JspException
```

```
     */
    public static CommandTagException[] getExceptionsByType(PageContext
pageContext, String type) throws JspException {
        if (type == null) {
            return new CommandTagException[0];
        }
        try {
            Class cls = Class.forName(type);
            return getExceptionsByClass(pageContext, cls);
        }
        catch (ClassNotFoundException e) {
            throw new JspException("Could not load class "+type+" in
TagSupportFramework.getExceptionsByType()");
        }
    }

    /**
     * Examines the page exception stack and returns an
     * array of exceptions which match the class passed in.
     *
     * @param pageContext
     *                JSP page context used to retrieve the exception stack
     * @param cls     The class of exception to search for in the
     *                exception stack
     * @return an array of CommandTagExceptions.  Will return a zero-
     *         length array if no exceptions in the stack matched the
     *         given class or if the class parameter is null.
     * @exception JspException
     */
    public static CommandTagException[] getExceptionsByClass(PageContext
pageContext, Class cls) {
        Vector exceptionHold = (Vector)
pageContext.getRequest().getAttribute(TagConstants.EXCEPTION_HOLD_NAME);
        if (exceptionHold == null) {
            return new CommandTagException[0];
        }

        Vector matches = new Vector(0, 1);
        for (Enumeration e = exceptionHold.elements();
e.hasMoreElements();) {
            Object o = e.nextElement();
            if (cls.isInstance(o)) {
                matches.addElement(o);
            }
        }
        CommandTagException[] result = new
CommandTagException[matches.size()];
        matches.copyInto(result);
        return result;
    }

    /**
     * Find out how many exceptions are currently stored
     * in the page exception stack
     *
     * @param pageContext
     *                JSP page context object used to retrieve the stack
     * @return the size of the page exception stack
```

```
     */
    public static int getExceptionCount(PageContext pageContext) {
        Vector exceptionHold = (Vector)
pageContext.getRequest().getAttribute(TagConstants.EXCEPTION_HOLD_NAME);
        if (exceptionHold==null)
            return 0;
        return exceptionHold.size();
    }

    /**
     * If the exception hold contains any exceptions,
     * return true, otherwise return false.
     *
     * @param pageContext
     * @return
     */
    public static boolean haveExceptionErrors(PageContext pageContext) {
        CommandTagException[] exceptions =
getExceptionsByClass(pageContext,SuccessException.class);
        if (getExceptionCount(pageContext) > exceptions.length) {
            return true;
        }
        return false;
    }


}
```

```
/*
CommandTagExample.java is an example of how to write a simple command
tag class which extends AbstractCommandTag.  By virtue of extending
AbstractCommandTag, CommandTagExample can do page routing and error
handling, as illustrated in the source code.
*/


import java.io.*;

public class CommandTagExample extends AbstractCommandTag {

    /**
     * This command tag opens a server-side file and writes
     * the text "hello, world!".
     *
     * @param pageContext
     * @exception CommandTagException
     */
    protected void execute(PageContext pageContext) throws
CommandTagException {
        try {
            File logfile = new File("/var/tmp/example.log");
            FileWriter writer = new FileWriter(logfile);
            writer.write("hello, world!");
            writer.close();
        }
        catch (FileNotFoundException e) {
            /**
             * For example purposes, this error is determined to be
recoverable,
             * so the error is added to the list of recoverable errors for
the
             * current page.  The AbstractCommandTag will now route the
user based
             * based on the onFailure tag attribute.
             */
            TagFrameworkSupport.throwToPage(pageContext, new
CommandTagException("Could not find example log file"));
        }
        catch (IOException e) {
            /**
             * For example purposes, this error is determined to be
critical, so
             * when it occurs, page execution should stop.  This is
indicated by
             * actually throwing a Java exception rather than calling the
throwToPage()
             * method in TagFrameworkSupport.
             */
            throw new CommandTagException("Error writing to the example log
file!");
        }
    }
}
```

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division

Application deficiencies found during scanning:

☐ Page(s)_____of_____ were not present for scanning.
(Document title)

☐ Page(s)_____of_____ were not present for scanning.
(Document title)

* Appendix out of order numbers.

☐ ***Scanned copy is best available.***